Table of Contents

Executive Summary

The purpose of the project

This project focuses on the application and evaluation of 2 local search optimization techniques, **Simulated Annealing (SA)** and **Genetic Algorithm (GA)**, to solve the problem of Neural Network pruning. Its primary objective was to reduce model complexity and enhance computational efficiency without significantly compromising performance. This task can enable models to run more efficiently with reduced memory and computational demands, making them suitable for deployment on resource-constrained devices like edge computing systems.

The problem tackled: Pruning is a crucial technique in Deep Learning model optimization, as it reduces the number of weights, leading to smaller model sizes, faster inference times, and lower memory usage. Traditional pruning techniques rely on predefined rules or assumptions about which weights to prune. This project will "bypass" these limitations by leveraging local search algorithms to dynamically learn the best pruning masks.

AI technique used

Simulated Annealing and **Genetic Algorithm** were experimented here to solve this optimization problem. They will be used as a means of automatically discovering best pruning masks that eliminate unnecessary weights without making any rule-based assumptions regarding weight importance:

1. **Simulated Annealing (SA)**: A probabilistic local search method inspired by the physics of annealing in metallurgy, which explores the pruning mask space by accepting suboptimal solutions at higher "temperatures" to escape local minima, gradually refining the solution as the temperature cools. By gradually lowering this "temperature", it controls the probability of accepting worse solutions as the algorithm refines its pruning masks.
2. **Genetic Algorithm (GA)**: An evolutionary algorithm simulating evolutionary processes such as natural selection, crossover, and mutation. It maintains a population of possible candidate pruning solutions (chromosomes) and evolves them over several generations to discover optimal pruning masks. **GA** is particularly effective at exploring a larger solution space due to its population-based nature, allowing for more aggressive pruning strategies.

LeNet, a convolutional Neural Network (CNN), was used as the baseline architecture in this project. With a **layer-wise pruning** strategy and adaptive parameters, the goal of 2 above algorithms is to prune a **LeNet-5** model trained on the **MNIST** dataset, which achieved a baseline accuracy of **97.23%** with **61,470** weights. The challenge was to achieve maximum **sparsity** while retaining the model's **loss** by optimizing an objective function penalizing poor performance.

Major findings

1. **Pruning Effectiveness**: **SA** achieved slightly higher **43.84%** sparsity with **87.44%** accuracy (drop **9.8%** from the baseline), while **GA** reached **41.33%** sparsity with **89.02%** accuracy, demonstrating that both methods can significantly reduce model size while maintaining reasonable performance.
2. **Efficiency**: **SA** proved to be faster than **GA**, requiring less than half the execution time. It completed the pruning process in *11 minutes 43 seconds*, whereas **GA** required *27 minutes 46 seconds*. This makes **SA** more suitable for applications where time efficiency is critical.
3. **Convergence Behaviour**: **SA** exhibited rapid initial improvement, particularly beneficial for quick results, while **GA** showed more gradual, steady improvements across generations.

Conclusion and recommendations

Overall, the project successfully demonstrated that both SA and GA as powerful tools for automated model pruning, with **SA** showing a slight edge in balancing sparsity, accuracy/loss and execution time. The choice of method should depend on specific requirements of the deployment environment:

- In resource-constrained environments or cases where aggressive pruning is desired, **SA** can provide faster execution and moderate pruning with higher sparsity, making it viable for lightweight models like **LeNet-5**.
- For large-scale models where maintaining accuracy/loss is key, the **GA** is the recommended technique. By offering deeper exploration of the solution space, **GA** may balance sparsity and accuracy/loss more effectively in these models compared to **SA**, though care must be taken to time constraints.
- A hybrid approach combining the rapid convergence of **SA** with the diverse exploration of **GA** could be considered: using **SA** for an initial rapid pruning and then refining the pruning masks with the **GA** for accuracy retention.
- Implementing and fine-tuning or iteratively pruning to potentially lead to superior results (Song et al., 2015) when applying these algorithms to new pruning tasks.

Introduction

What was the problem and its context?

Neural Networks is the foundation of modern AI. They offer remarkable performance on various tasks, from image classification to natural language processing. However, this performance often comes at the cost of a large number of weights, resulting in high computational resources and large memory footprints. This cost between model performance and efficiency necessitates effective methods to reduce the model complexity while still maintaining high metrics like accuracy for f1-scores. Model pruning, a process of removing redundant or less important weights from a model while maintaining performance, is one of the key optimization techniques to address this issue.

Why was it a problem?

The large number of weights in modern Neural Networks can lead to inefficiencies in both computation and storage. These inefficiencies become particularly problematic in scenarios where computational power and memory are constrained, such as edge devices, mobile platforms, IoT applications, or large-scale production systems. In such environments, reducing the size/complexity and computational requirements of the models while maintaining their performance is crucial for practical deployment. Moreover, large networks may even be inefficient as smaller ones due to an idea known as *Lottery Ticket Hypothesis* (Frankle & Carbin, 2019), suggesting that an over-parameterized network may contains a smaller network (winning ticket), which can achieve comparable performance when trained from scratch. This means that within bigger network, there is a more compact and effective version of it that can perform just as well if trained on its own.

Why was the project necessary and how was the problem solved?

Traditional pruning rule-based methods require prior knowledge of which weights to prune (e.g., remove weights below a threshold) (Li et al., 2020), making them less adaptive and not scalable for more complex architectures. This limits the pruning process, especially in models where the importance of weights may not follow simple patterns. Reducing the model's size and computational demands without relying on heuristic-driven techniques requires an approach that can dynamically adapt to the network's structure and data distribution.

As models are increasingly deployed in real-time applications (often on devices with limited computational capabilities), this project was necessary to explore more automated and intelligent approaches to network pruning by leveraging local search optimization techniques, specifically **Simulated Annealing (SA)** and **Genetic Algorithms (GA)**, to intelligently learn which weights can be pruned with minimal impact on accuracy. These optimization algorithms were chosen because of their flexibility in exploring large search spaces and their ability to find optimal or near-optimal solutions in non-convex spaces without relying on assumptions about weight importance. Unlike traditional methods, **SA** and **GA** allow for a dynamic search for the best pruning masks, learning from the model's structure and performance during pruning, and thus making the process more scalable and generalizable to different architectures.

In this project, the specific problem will be revolved around optimizing a simple Neural Network (**LeNet**) trained on the **MNIST** dataset through 2 above automated pruning techniques. Here, **SA** and **GA** were implemented with adaptive parameters to **prune each layer** of this **LeNet** model, effectively navigating the complex search space of possible pruning configurations. These algorithms evaluated the performance of each pruning configuration using an objective function aiming to **maximize sparsity** while **minimizing the loss**. The results of each method were then compared in terms of sparsity achieved, accuracy and loss retained, and execution time, providing insights into pros & cons of each approach.

GOAL: Develop automated pruning methods to reduce the model's complexity and computational requirements by pruning weights across different layers, leading to a sparse model that retains as much of its original accuracy/loss as possible, offering solutions to prune neural networks without prior assumptions knowledge of weight importance.

Report Body

The search for the optimal pruning mask is treated as an optimization problem. For a fair comparison between **Simulated Annealing** and **Genetic Algorithms**, both techniques will use the same sophisticated objective function.

```python
def calculate_objective(self):
    loss, metrics = self.pruned_model.evaluate(self.baseline.data.x_test, self.baseline.data.y_test, verbose=0)
    sparsity = np.mean([np.mean(mask == 0) for mask in self.masks if mask is not None])
    loss_diff = (self.baseline.loss - loss) ** 2 if loss < self.max_loss else self.max_loss_penalty
    cost = loss_diff + 1 / (sparsity + 1e-8)
    return {'cost': cost, 'metrics': metrics, 'loss': loss, 'sparsity': sparsity}
```

This function evaluates the costs of pruning masks based on 2 key criteria: loss difference (performance degradation of the pruned model compared to the original model) and effective sparsity (the level of sparsity achieved in each layer), along with a small *epsilon* constant (10^{-8}) to avoid division by 0. This cost function is expressed as:

$$cost = loss\_diff + \frac{1}{sparsity + \varepsilon} \ (See\ Appendix\ A\ for\ more\ details)$$

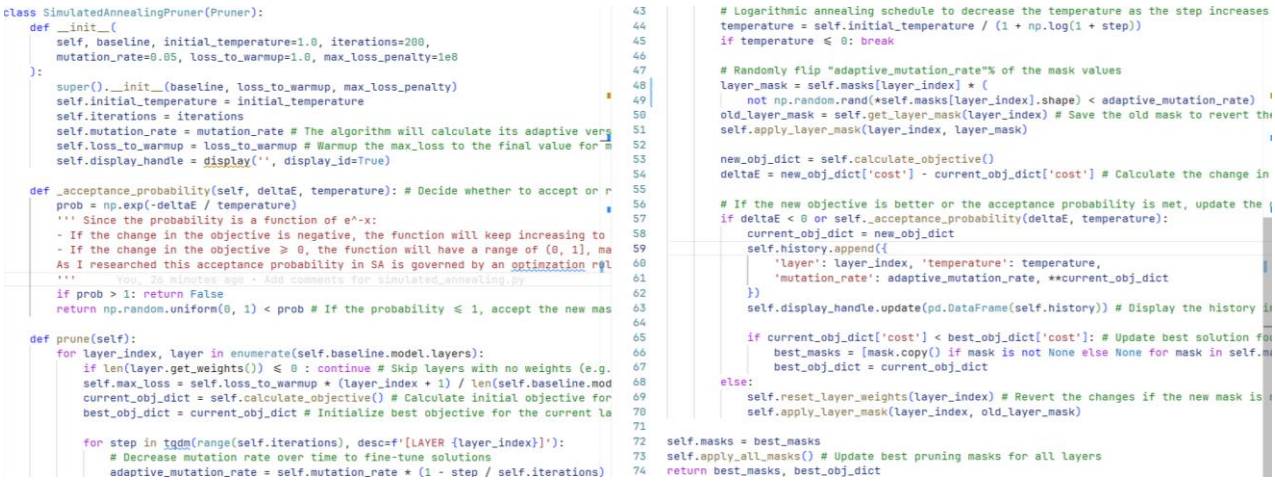
This formulation encouraged the algorithm to find a balance between minimizing loss and maximizing sparsity, with a high penalty for excessive loss (exceed the **max_loss**). The loss difference (**loss_diff**) is also squared to penalize high loss more than low loss.

Methods/AI techniques used

1. Simulated Annealing (SA)

This probabilistic technique is inspired by the annealing process in metallurgy, where metal gets heated and slowly cooled to minimize its energy state, remove defects, and reach a stable configuration (Shachar, 2024). In this context of Neural Network pruning, the goal of **SA** is to find an optimal configuration of these masks such that the network becomes sparse without sacrificing too much accuracy. It is applied to iteratively explore the vast combinatorial search space of optimal pruning masks and refine the current ones applied to the network. In my implementation, I focused on SA layer-wise, allowing for an optimization of each layer's pruning mask.

How Simulated Annealing Works for Pruning:



```
class SimulatedAnnealingPruner(Pruner):
    def __init__(
        self, baseline, initial_temperature=1.0, iterations=200,
        mutation_rate=0.05, loss_to_warmup=1.0, max_loss_penalty=1e8
    ):
        super().__init__(baseline, loss_to_warmup, max_loss_penalty)
        self.initial_temperature = initial_temperature
        self.iterations = iterations
        self.mutation_rate = mutation_rate # The algorithm will calculate its adaptive vers
        self.loss_to_warmup = loss_to_warmup # Warmup the max_loss to the final value for m
        self.display_handle = display('', display_id=True)

    def _acceptance_probability(self, deltaE, temperature): # Decide whether to accept or r
        prob = np.exp(-deltaE / temperature)
        ''' Since the probability is a function of e^-x:
        - If the change in the objective is negative, the function will keep increasing to
        - If the change in the objective ⩾ 0, the function will have a range of (0, 1], ma
        As I researched this acceptance probability in SA is governed by an optimization rul
        '''      You, 20 minutes ago • Add comments for simulated_annealing.py
        if prob > 1: return False
        return np.random.uniform(0, 1) < prob # If the probability ⩽ 1, accept the new mas

    def prune(self):
        for layer_index, layer in enumerate(self.baseline.model.layers):
            if len(layer.get_weights()) ⩽ 0 : continue # Skip layers with no weights (e.g.
            self.max_loss = self.loss_to_warmup * (layer_index + 1) / len(self.baseline.mod
            current_obj_dict = self.calculate_objective() # Calculate initial objective for
            best_obj_dict = current_obj_dict # Initialize best objective for the current la

            for step in tqdm(range(self.iterations), desc=f'[LAYER {layer_index}]'):
                # Decrease mutation rate over time to fine-tune solutions
                adaptive_mutation_rate = self.mutation_rate * (1 - step / self.iterations)
                # Logarithmic annealing schedule to decrease the temperature as the step increases
                temperature = self.initial_temperature / (1 + np.log(1 + step))
                if temperature ⩽ 0: break

                # Randomly flip "adaptive_mutation_rate"% of the mask values
                layer_mask = self.masks[layer_index] * (
                    not np.random.rand(*self.masks[layer_index].shape) < adaptive_mutation_rate)
                old_layer_mask = self.get_layer_mask(layer_index) # Save the old mask to revert th
                self.apply_layer_mask(layer_index, layer_mask)

                new_obj_dict = self.calculate_objective()
                deltaE = new_obj_dict['cost'] - current_obj_dict['cost'] # Calculate the change in

                # If the new objective is better or the acceptance probability is met, update the
                if deltaE < 0 or self._acceptance_probability(deltaE, temperature):
                    current_obj_dict = new_obj_dict
                    self.history.append({
                        'layer': layer_index, 'temperature': temperature,
                        'mutation_rate': adaptive_mutation_rate, **current_obj_dict
                    })
                    self.display_handle.update(pd.DataFrame(self.history)) # Display the history i

                    if current_obj_dict['cost'] < best_obj_dict['cost']: # Update best solution fo
                        best_masks = [mask.copy() if mask is not None else None for mask in self.m
                        best_obj_dict = current_obj_dict
                else:
                    self.reset_layer_weights(layer_index) # Revert the changes if the new mask is
                    self.apply_layer_mask(layer_index, old_layer_mask)

        self.masks = best_masks
        self.apply_all_masks() # Update best pruning masks for all layers
        return best_masks, best_obj_dict
```

The search space here is defined by all possible combinations of pruning masks across the layers of the network. Each mask is a binary matrix indicating whether a particular weight in a given layer is kept or pruned. **SA** will begin by exploring this solution space more freely, with a high probability of accepting worse solutions (i.e., solutions that result in higher cost). As the temperature decreases, the algorithm becomes more conservative, focusing on refining better solutions. Specifically, it initializes the pruning mask and sets the ***initial_temperature*** to a high value, allowing for exploration of different mask configurations, then gradually "cools down" this temperature as it explores the solution space. This temperature schedule follows a logarithmic decay $T = T_0/(1 + \log(1+step))$ (**line 36**). It ensured a slow cooling or a gradual transition from more exploration early in the pruning process and greater exploitation as the algorithm progressed.

At each iteration, this temperature decreases, reducing the probability of accepting worse solutions. Then, **SA** proposes a neighbor state (a new candidate) by flipping a subset of bits in the current pruning mask with a probability determined by an adaptive mutation rate. This enables the algorithm to explore different configurations of pruned and retained weights (**line 40**). The ***adaptive_mutation_rate*** applied here is one of my enhancements compared to the original version of **SA** to improve its effectiveness in the context of Neural Network pruning, ensuring a balance between exploration in early stages and exploitation in later stages. The algorithm will then evaluate whether this new configuration/solution improves the overall performance or sparsity by applying the above sophisticated cost function that balances loss and sparsity. If the new solution improves this cost, it is accepted. If not, it still may be accepted with a probability proportional to the temperature that decreases as the algorithm progresses:

$$P_{acceptance}(new\ solution) = \begin{cases} 1, & \Delta E < 0 \\ e^{\frac{-\Delta E}{T}}, & \Delta E \geq 0 \end{cases}, with\ \Delta E = cost_{new\ solution} - cost_{current\ solution}$$

ΔE represents the change in the cost function and T is the current temperature. This probabilistic acceptance criterion allows the algorithm to occasionally accept worse solutions, helping it escape local optima. As the temperature cools, the algorithm becomes more selective, converging towards a near-optimal pruning configuration where sparsity is maximized, and loss is maintained. According to my research, this is governed by a rule called *Metropolis criterion*.

Relevance and Effectiveness:

SA is particularly suited for problems with large and complex search space, like this Neural Network pruning. This is because it can escape local minima by accepting temporary increases in the objective function, a crucial property given

the non-convex nature of the search space in Neural Network pruning, making it an effective tool for exploring a wide range of potential pruning configurations. By progressively narrowing the search space as the temperature cools with my sophisticated objective function, **SA** is able to find an optimal balance between maximizing model sparsity and minimizing the loss. In this problem, **SA** helps to automate the pruning process by iteratively improving pruning masks without relying on predefined rules. Moreover, my adaptive terms further refine the search, allowing for more aggressive exploration during the early stages and fine-tuning in the later stages.

2. Genetic Algorithm (GA)

This method is inspired by the process of natural selection, where populations evolve over time through natural operations such as selection, crossover, and mutation. It simulates evolution through generations, combining the fittest solutions (chromosomes) from one generation to produce offspring for the next generation. In Neural Network pruning, **GA** is used to evolve populations of pruning masks, with the goal of discovering masks that yield high sparsity while preserving accuracy. My enhancements in this **GA** implementation include ***elitism*** to preserve the best solutions across generations, and an ***adaptive_mutation_rate*** like the **SA** implementation. The objective of **GA** is also to find an optimal pruning mask that reduces the complexity of the network without sacrificing loss. The fitness of each solution (i.e., pruning mask) is evaluated based on its ability to maintain low loss while pruning unnecessary weights.

How Genetic Algorithms Work for Pruning:

```
class Chromosome:
    def __init__(self, pruner, layer_index, layer_mask=None, init_rate=0.1, disable_prune=F
        self.pruner = deepcopy(pruner) # Create a copy of the pruner as it will be modified
        self.init_rate = init_rate # Initial mutation rate, only for initializing layer mas
        self.layer_index = layer_index
        self.layer_mask = self._initialize_layer_mask() if layer_mask is None else layer_ma
        self.obj_dict = {}

        # The cost for each Chromosome is automatically calculated when it is created
        if not disable_prune: # Avoid applying mask in Crossover to make it faster
            self.pruner.apply_layer_mask(self.layer_index, self.layer_mask)
            self.obj_dict = self.pruner.calculate_objective()

    def _initialize_layer_mask(self):    You, 4 days ago • Add genetic algorithm based pr
        layer = self.pruner.pruned_model.layers[self.layer_index]
        return np.random.choice([0, 1], size=layer.get_weights()[0].shape, p=[self.init_rat

    def crossover(self, other: 'Chromosome', crossover_rate=0.9) -> Tuple['Chromosome', 'Ch
        # Create 2 new offspring chromosomes from 2 parents (self and other)
        if np.random.random() < crossover_rate:
            # Codes below were cited from line 418-420 of
            # https://github.com/Ruturaj-Godse/automated-model-pruning-using-genetic-algori
            # https://github.com/Ruturaj-Godse/automated-model-pruning-using-genetic-algorithm/
            crossover_point = np.random.randint(0, self.layer_mask.size - 1) # Single-point cro
            mask1_flat, mask2_flat = self.layer_mask.flatten(), other.layer_mask.flatten() # Fl
            child1_mask = np.concatenate([mask1_flat[:crossover_point], mask2_flat[crossover_po
            child2_mask = np.concatenate([mask2_flat[:crossover_point], mask1_flat[crossover_po
            child1_mask = child1_mask.reshape(self.layer_mask.shape)
            child2_mask = child2_mask.reshape(other.layer_mask.shape)
            return (
                Chromosome(self.pruner, self.layer_index, child1_mask, disable_prune=True),
                Chromosome(self.pruner, self.layer_index, child2_mask, disable_prune=True)
            )
        return deepcopy(self), deepcopy(other) # Return deep copies of parents if the crossover

    def mutate(self, mutation_rate=0.05):
        # If mutation rate is less than random rate, then randomly change
        # the genes of the layer mask with that given mutation rate
        self.layer_mask = np.array([
            np.random.choice([0, 1], p=[1-mutation_rate, mutation_rate])
            if np.random.random() < mutation_rate else gene for gene in self.layer_mask.flatte
        ]).reshape(self.layer_mask.shape)
        self.pruner.apply_layer_mask(self.layer_index, self.layer_mask)
        self.obj_dict = self.pruner.calculate_objective()
```

```
You, 32 minutes ago | 1 author (You)
class GeneticAlgorithmPruner(Pruner):
    def __init__(    You, 4 days ago • Add genetic algorithm based pruning implementat
            self, baseline, num_generations=15, population_size=10,
            tournament_size=5, crossover_rate=0.9, mutation_rate=0.1,
            elite_size=2, loss_to_warmup=1.0, max_loss_penalty=1e8
    ):
        super().__init__(baseline, loss_to_warmup, max_loss_penalty)
        self.num_generations = num_generations # Number of generations to evolve in each la
        self.population_size = population_size # Number of Chromosomes in each generation
        self.tournament_size = tournament_size # Number of Chromosomes to sample in each to
        self.crossover_rate = crossover_rate # Probability of applying crossover, higher va
        self.mutation_rate = mutation_rate # The algorithm will calculate its adaptive vers
        self.elite_size = elite_size # Number of best Chromosomes to keep in each generatio
        self.loss_to_warmup = loss_to_warmup # The maximum loss to reach in the last layer


    def initialize_layer_population(self, layer_index) -> List[Chromosome]:
        # Keep initializing the population until at least 1 chromosome has a cost less than
        # It will reduce the randomness of 0 in masks to keep the performance close to the
        init_rate = 1.0
        while True:
            init_rate *= self.mutation_rate
            population = [ # Initialize population with random pruning masks
                Chromosome(self, layer_index, init_rate=init_rate)
                for _ in tqdm(range(self.population_size), desc=f'[LAYER {layer_index}] Ini
            ]
            if all(chromosome.obj_dict['cost'] > self.max_loss_penalty for chromosome in po
            break
        return population


    def tournament_select(self, population: List[Chromosome]) -> List[Chromosome]:
        selected_chromosomes = []
        for _ in range(2): # Select 2 parents
            # Randomly sample chromosomes from the population and select the best one,
            # replace=False avoids selecting the same chromosome twice
            tournament = np.random.choice(population, size=self.tournament_size, replace=Fa
            winner = min(tournament, key=lambda chromosome: chromosome.obj_dict['cost'])
            selected_chromosomes.append(winner)
        return selected_chromosomes
```

```
def evolve_layer_population(self, layer_index, population: List[Chromosome]) -> Chromosome:
    best_chromosome = None
    for generation in range(self.num_generations):
        # Apply mutation by decreasing mutation rate over time to fine-tune solutions
        adaptive_mutation_rate = self.mutation_rate * (1 -generation / self.num_generations

        # Sort population by cost to push best solutions (lowest cost) to the top for elite
        population.sort(key=lambda chromosome: chromosome.obj_dict['cost'])
        new_population = population[:self.elite_size] # Apply elitism

        while True: # Selection, Crossover, and Mutation
            pbar = tqdm(
                total=self.population_size - self.elite_size, initial=self.elite_size,
                desc=f'[LAYER {layer_index}] Evolving Generation {generation}'
            )
            while len(new_population) < self.population_size: # Generate new population
                parent1, parent2 = self.tournament_select(population)
                child1, child2 = parent1.crossover(parent2, self.crossover_rate)
                child1.mutate(adaptive_mutation_rate)
                child2.mutate(adaptive_mutation_rate)
                new_population.extend([child1, child2]) # Add offspring to the new populati
                pbar.update(2)

            pbar.close()
            population = new_population # Replace population with new generation
            new_population = [] # Reset new population for the next generation

            # Again, keep forming a new population until at least 1 chromosome has a cost l
            if all(chromosome.obj_dict['cost'] > self.max_loss_penalty for chromosome in po
            break

        current_best_chromosome = min(population, key=lambda chromosome: chromosome.obj_dic
        if best_chromosome is None or current_best_chromosome.obj_dict['cost'] < best_chrom
            best_chromosome = current_best_chromosome # Update the best solution (best pru

        self.history.append({
            'layer': layer_index, 'generation': generation,
            'mutation_rate': adaptive_mutation_rate, **current_best_chromosome.obj_dict,
        })
        clear_output(wait=True)
        display(pd.DataFrame(self.history))
    return best_chromosome # Best layer mask found
```

In this context, each individual/Chromosome in the population is represented by a set of potential pruning masks across all layers. Like in **SA**, a mask is a binary matrix indicating whether a weight is retained or pruned. When a population of individuals/Chromosomes (pruning mask configurations) is initialized, each individual is randomly generated with an initial ***mutation_rate*** or probability of retaining or pruning weights to encourage initial performance. The population size and initialization strategy directly impact the diversity of the search, and each generation will apply genetic operators to create new offspring:

- **Selection**: Selection mechanism is performed using tournament selection, where the Chromosomes with higher fitness (better pruning masks) have a higher tendency of being selected to reproduce. 2 chosen Chromosomes from the population will serve as parents for crossover and for the next generation. The fitness of each chromosome is evaluated in the same way as the cost function in **SA**.

- **Crossover**: During crossover, 2 parent Chromosomes exchange segments of their pruning masks to produce offspring, applying single-point crossover with a probability of 0.9. The offspring inherit characteristics from both parents, allowing the algorithm to combine effective pruning strategies from different individuals.
- **Mutation**: Mutation introduces small random changes to the offspring's pruning masks, randomizing some bits to explore new pruning configurations of the search space with an adaptive rate that decreases over generations, allowing for fine-tuning of solutions in later stages. This helps maintain genetic diversity and prevents premature convergence on suboptimal solutions.
- **Elitism**: The use of elitism here ensures the best-performing Chromosomes from the current generation are preserved and carried over to the next generation to ensure that high-quality solutions are not lost.

Relevance and Effectiveness:

In the context of Neural Network pruning, **Genetic Algorithm** is also particularly effective for large search spaces, discovering diverse pruning strategies that yield high model sparsity. The combination of crossover and mutation allows **GA** to explore a broad range of potential solutions while maintaining diversity in the population:

- The use of crossover allows the algorithm to combine the best features of different solutions, accelerating convergence towards optimal pruning configurations.
- The mutation mechanism adds diversity here, "preventing premature convergence to suboptimal solutions" (Shachar, 2024).

This evolutionary nature of the algorithm allows for continual improvement towards optimal or near-optimal pruning configurations over generations, leading to practical pruning masks that maximize sparsity without excessively compromising accuracy.

3. Layer-wise with Adaptive terms

A unique feature in my implementation is the **layer-wise** evolution strategy, which allows for focused optimization of each layer's pruning mask. Both techniques are applied sequentially to each layer with a novel "warmup" strategy for adaptive maximum allowable loss threshold that gradually increases as progressing through the layers:

$$max\_loss(layer\_index) = loss\_to\_warmup * \frac{layer\_index + 1}{Total\ Layers}$$

This approach allows for more aggressive pruning in the earlier layers while being more conservative in deeper layers, which are typically more sensitive to pruning. Both techniques are also applied with ***adaptive_mutation_rate*** to enhance their effectiveness over time:

$$adaptive\_mutation\_rate_{Simulated\ Annealing} = mutation\_rate - \frac{1 - step}{Total\ iterations}$$

$$adaptive\_mutation\_rate_{Genetic\ Algorithm} = mutation\_rate - \frac{1 - generation}{Total\ generations}$$

These adaptive rates allow for broader exploration in early stages and finer tuning towards the end, contributing to the algorithms' ability to find good pruning configurations.

Implementation of AI techniques

1. Data source and Data pre-processing

The primary data source used for this experiment is the well-known **MNIST** dataset, a standard in the machine learning community. It contains **70,000 grayscale images** of handwritten digits, split into **60,000 training** images and **10,000 test** images (Lecun et al., 1998, p. 10). It contains **10 classes**, representing 10 digits from 0 to 9. Each image has a size of **28x28 pixels** with values ranging from 0 to 255. MNIST is an excellent benchmark for my study due to its manageable size and widespread use for evaluating neural network performance, allowing for rapid experimentation and iteration. Here, the **MNIST** dataset is preloaded using TensorFlow's built-in dataset loader, **mnist.load_data()**.

```
class MNIST:
    def __init__(self):
        # Load MNIST dataset
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
        self.y_train, self.y_test = y_train, y_test
    You, 4 days ago • Add MNIST dataset handling, LeNet model impleme…
        # Add channel dimmension and normalize pixel values between 0 and 1
        self.x_train = x_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
        self.x_test = x_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
        self.input_shape = self.x_train.shape[1:]
        self.num_classes = len(set(self.y_train))
```

```
Training data shape: (60000, 28, 28, 1)
Training labels shape: (60000,)
Test data shape: (10000, 28, 28, 1)
Test labels shape: (10000,)
Input shape: (28, 28, 1)
Number of classes: 10
```

While the dataset is already well-processed, a few key preprocessing steps were taken to ensure optimal performance for the **LeNet** model:

- **Reshaping**: The dataset was reshaped to include a channel dimension of 1 to ensure that input images are compatible with the expected input shape of the first CNN layer in the **LeNet** architecture.
- **Normalization**: All pixel values in the images were scaled to the range [0, 1] by dividing the raw pixel values by 255.0. This ensures that the inputs to the neural network are within a standardized range, accelerating model convergence by preventing large gradients during backpropagation.

This dataset will further split into the training set by 20% during training to create the validation set. This split allows for proper model evaluation and helps prevent overfitting.

2. Baseline Training for LeNet

The **LeNet** architecture (Lecun et al., 1998) is a classic convolutional neural network (CNN) widely used for image classification tasks, consisting of:

- **2 Convolutional layers**: Extracting spatial features from input images.
- **2 Average Pooling layers**: Reducing dimensionality while retaining important features.
- **3 Fully connected layers**: Making predictions based on the learned features.

Layer (type)	Output Shape	Param #	Traina…
Conv1 (Conv2D)	(None, 28, 28, 6)	156	Y
AvgPool1 (AveragePooling2D)	(None, 14, 14, 6)	0	-
Conv2 (Conv2D)	(None, 10, 10, 16)	2,416	Y
AvgPool2 (AveragePooling2D)	(None, 5, 5, 16)	0	-
Flatten (Flatten)	(None, 400)	0	-
FC1 (Dense)	(None, 120)	48,120	Y
FC2 (Dense)	(None, 84)	10,164	Y
Output (Dense)	(None, 10)	850	Y

```
Model name: LeNet
Total weights: 61470
Test loss: 0.0768
Test metrics: 0.9753999710083008
Prunable layers: [0, 2, 5, 6, 7]
```

All code is executed in a **Google Colab PRO** environment, which provides High-RAM and GPU acceleration, essential for the timely execution of training and evaluating **LeNet**. The experiment is conducted using TensorFlow as the primary deep learning framework. For baseline training on **MNIST**, this model is compiled using Stochastic Gradient Descent (SGD) with a learning rate of *0.1* (chosen for its simplicity and efficiency in this experimental setting), and the loss function used is *sparse_categorical_crossentropy*, ideal for multi-class classification tasks like digit recognition. This baseline model was trained for *50* epochs with a batch size of *128* and a *20%* validation split. The baseline setting achieves an accuracy of **97.23%** on the test set, with a loss of **0.0895**. The model contains **61,470** total weights, providing a potential opportunity for pruning.

LeNet was proven effective for **MNIST**, but like most Neural Networks, it is prone to over-parameterization, meaning it often contains many redundant or insignificant weights that do not contribute meaningfully to its performance. In the context of **LeNet**, pruning without predefined assumptions is a non-trivial task. Layers such as fully connected layers contain a high number of parameters, which are often highly redundant. However, convolutional layers, which capture spatial patterns, may require more careful pruning to maintain performance. Finding an optimal pruning mask across layers and ensuring model performance can present a significant optimization challenge.

3. Experiment Setup and Parameters Used

SimulatedAnnealingPruner was applied to prune the trained **LeNet** with the goal to optimize the pruning mask for each layer to achieve higher sparsity (fewer non-zero weights) while maintaining the test loss. The experiment involved applying an iterative process of mutating the pruning masks, evaluating the model's performance, and accepting or rejecting solutions based on an acceptance probability that depended on both the temperature and the difference in the objective function between solutions. Its parameters used include:

- **Initial_temperature**: *1.0* — This high starting temperature allowed for greater exploration by accepting solutions with higher cost early in the process.
- **iterations**: *200* — For each layer, the algorithm performed 200 iterations of pruning mask updates.

GeneticAlgorithmPruner, on the other hand, was applied using a population-based approach. In each generation, the algorithm evolved a population of chromosomes (pruning masks) through selection, crossover, and mutation. The goal was to optimize the pruning masks for each layer, striking a balance between model sparsity and accuracy. **GA** used both exploration and exploitation through random mutations and recombination of masks from top-performing chromosomes. Its parameters used include

- **Number of Generations**: *20* — The population evolved over 20 generations.
- **Population Size**: *12* — Each generation consisted of 12 Chromosomes.

- **Tournament Size**: *5* — In each generation, parent chromosomes were selected using a tournament selection strategy involving *5* randomly chosen chromosomes.
- **Crossover Rate**: *0.9* — With a 90% probability, two selected parents would recombine their pruning masks. This high crossover rate encourages exploration of new solutions
- **Elite Size**: *2* — The top 2 Chromosomes from each generation were directly carried over to the next generation to preserve good solutions.

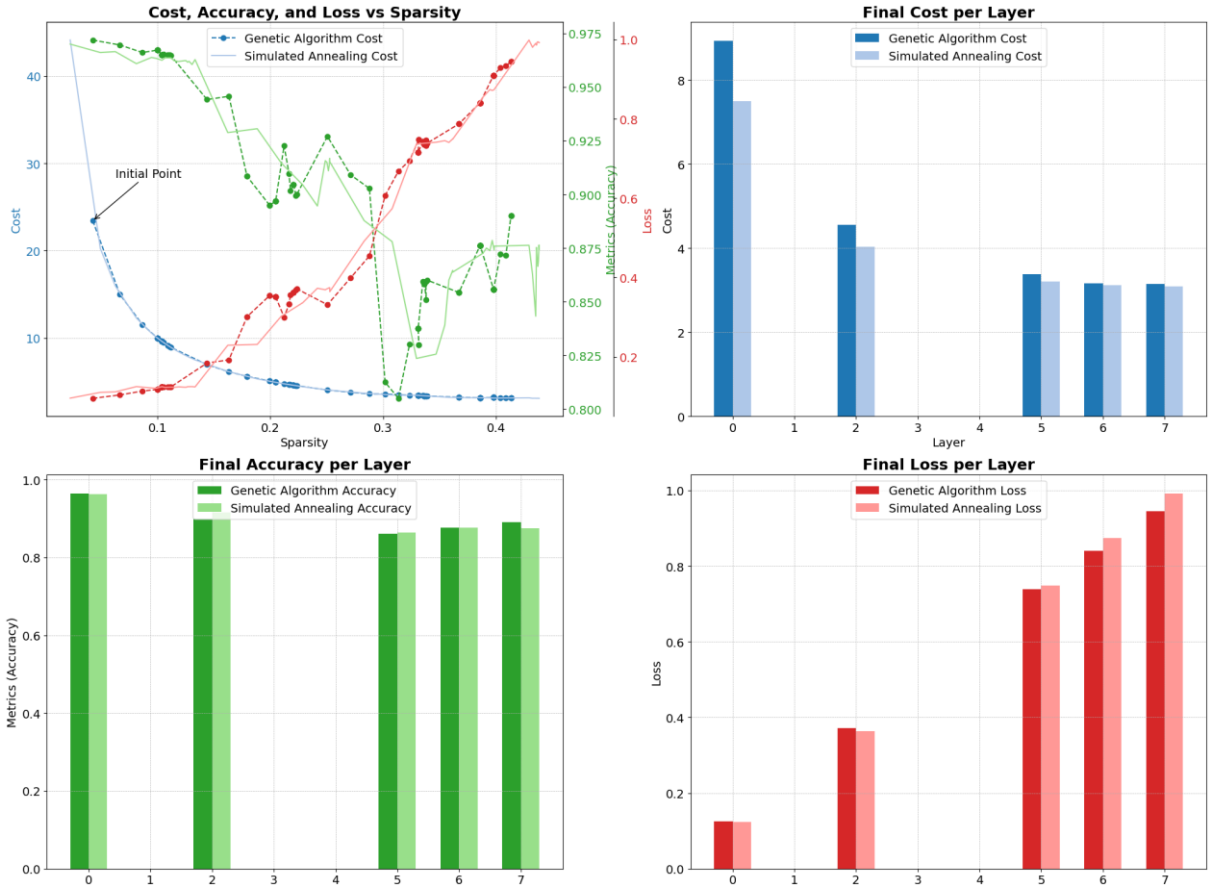
Both **SA** and **GA** will use an adaptive maximum loss threshold that increases for deeper layers, allowing for more aggressive pruning in later layers (**loss_to_warmup**: *1.0*) while preserving critical features in earlier layers. They also employed a high penalty (**max_loss_penalty**: 10^8) to discourage overly aggressive pruning that exceeds the allowed threshold and degrades accuracy. Moreover, to strengthen a fair comparison, I employed these following on purpose:

- Initial **mutation_rate** was set to *0.15*, meaning 15% of the weights in the pruning mask could change at the start of the process. Each algorithm will calculate its adaptive version to fine-tune solutions over time.
- The number of pruning loops in each algorithm will be **1000** in totals:

$$Total\ loop_{Simulated\ Annealing} = iterations * number\ of\ prunable\ layers = 200 * 5 = 1000$$

$$Total\ loop_{Genetic\ Algorithm} = num\_generations * (population\_size - elite\_size) * number\ of\ prunable\ layers$$
$$= 20 * (12 - 2) * 5 = 1000$$

4. Findings/Results



The experimental results from the **SA** and **GA** were compared based on several key values: cost, accuracy (metrics), loss, sparsity, and execution time. The results for each layer in the neural network were analyzed and compared in detail. Below, I discuss the findings from both pruning techniques and evaluate the quality of their solutions.

Layer	Simulated Annealing (SA)	Genetic Algorithm (GA)
Layer 0 *First convolutional layer*	Starting with a temperature of **1.0** and a mutation rate of **0.15**, the initial sparsity is **2.27%** with a cost of **44.12** and test loss of **0.1242** (close to the baseline with minimal accuracy loss of **97.0%**). As the temperature decreases, the sparsity increases steadily. The final result shows a sparsity of **13.33%**, with a cost of **7.50** and an accuracy of **96.28%**.	The initial sparsity is **4.27%**, with a cost of **23.43** and accuracy of **97.19%**. By generation **19th**, only **11.2%** of weights are removed, with a cost of **8.93**, test loss of **0.1244**, and accuracy of **96.46%**. This layer was pruned less aggressively compared to **SA**.

Layer 2 *Second convolutional layer*	For **Layer 2**, the initial sparsity is **16.23%** with a cost of **6.18** and accuracy of **92.88%**. After **200** iterations, the pruning process ends with a sparsity of **25.27%**, a cost of **4.03**, and an accuracy of **91.52%**.	The initial sparsity is **14.39%**, with an accuracy of **94.42%**. The pruning ends with a sparsity of **22.34%**, an accuracy of **89.9%,** and a cost of **4.55**.
Layer 5 *First fully connected layer*	Starting with a sparsity of **28.32%**, the final sparsity reaches **36.17%**, with the cost dropped significantly to **3.19**, potentially indicating that fully connected layers tend to have more redundant weights that can be pruned.	Initial sparsity starts at **25.06%**, with the final result achieving **33.79%** sparsity. Accuracy declines slightly to **86.01%**, but the cost improves to **3.37**. Similar to **SA**, this layer experienced high drop in cost, showing that fully connected layers may have more redundant parameters that can be pruned.
Layer 6 *Second hidden layer*	Final sparsity reaches **39.85%** and **43.83%**, respectively, with costs of around **3.1** and an accuracy ranging between **87.59%** and **87.44%**.	Sparsity increases from **36.72%** to **38.59%**, with a slight reduction in accuracy to **87.63%** and a cost of **3.16**.
Layer 7 *Output layer*		Final sparsity reaches **41.33%**, and accuracy stands at **89.02%** with a cost of **3.15**.

5. Quality of Solutions

Metrics	Simulated Annealing (SA)	Genetic Algorithm (GA)
Baseline accuracy	**97.23%**	
Final accuracy	87.44%	**89.02%**
Initial loss	0.0959	0.0953
Final loss	0.9923	**0.9442**
Sparsity gained	**0.4384**	0.4133
Time taken	**11 minutes 43 seconds**	27 minutes 46 seconds

Both **SA** and **GA** demonstrated significant model compression while maintaining reasonable accuracy. **SA** achieves a final sparsity of **43.84%** with a **9.79%-**point drop in accuracy, while **GA** reaches **41.33%** sparsity with **8.21%-**point drop. To quantify this trade-off, I use the following simple metric:

$$\text{Trade\_off Score} = \frac{Sparsity * Accuracy}{Baseline\ Accuracy} \rightarrow \begin{cases} Trade\_off_{Simulated\ Annealing} = \dfrac{(0.4384 * 0.8764)}{0.9723} = 0.3951 \\ Trade\_off_{Genetic\ Algorithm} = \dfrac{(0.4133 * 0.8902)}{0.9723} = 0.3784 \end{cases}$$

SA achieves a slightly better trade-off score, indicating that it may be more effective at balancing sparsity and accuracy. However, the difference is marginal, and both techniques prove effective for the task. They maintain a reasonable balance between pruning aggressiveness (increased sparsity) and maintaining accuracy/loss.

Comparisons of different AI techniques

1. Convergence behaviour with Cost function analysis

	Simulated Annealing (SA)	Genetic Algorithm (GA)
Cost	**SA** exhibited rapid initial improvement, with the cost function dropping from **44.12** to **7.50** in the **first layer** within **40 iterations**. Conversely, **GA** showed more stable convergence with smoother transitions between layers, demonstrating consistent improvement with the cost function decreasing from **23.44** to **8.93** in the **first layer** over **20 generations**.	
	The final costs across layers ranged from **7.50** in **layer 0** to **3.09** in **layer 7**. **SA**'s aggressive exploration of pruning masks allowed it to achieve low cost in later layers, especially where sparsity was high, but this was offset by a significant loss increase in some cases.	GA exhibited more consistent costs across layers, from **9.61 (layer 0)** to **3.15 (layer 7)**. **GA** uses **elitism** to ensure the best-performing chromosomes (masks) were retained, leading to more stable cost reduction over generations.
Pros	**SA** converges faster in terms of cost minimization and sparsity. The cost function stabilizes quickly after a few iterations, demonstrating that **SA** can reach near-optimal solutions efficiently.	While both methods have mechanisms to avoid local optima, **GA**'s population-based approach might provide a higher likelihood of discovering better configurations in complex and high-dimensional spaces.

Cons	While **SA** is fast and effective at finding a reasonable local minimum, it can sometimes get trapped in suboptimal solutions, especially when **temperature** decreases too quickly. This can limit the **SA**'s ability to explore better global solutions, particularly for deeper layers where pruning decisions are more complex.	**GA** requires more time to converge due to its computationally-expensive evolutionary nature The **population_size** and **num_generations** directly impact this execution time.

2. Loss on Test set and Accuracy retention

Accuracy retention is directly related to the model's **loss** after pruning. A pruning technique should aim to minimize the increase in **loss** while achieving high **sparsity**.

Simulated Annealing (SA)	Genetic Algorithm (GA)
The final test loss for **SA** varied across layers, from **0.1242 (layer 0)** to **0.9918 (layer 7)**. **SA** demonstrated more aggressive pruning, which led to a larger deviation in the loss, especially in later layers. The loss for **layer 7** (output layer) was **0.9918**, which is significantly (10 times) higher than the baseline, indicating that aggressive pruning came at the cost of reduced accuracy.	**GA** exhibited better accuracy retention, with losses from **0.1244 (layer 0)** to **0.9442** (**layer 7**). **GA**'s test losses were consistently **<** those of **SA** across most layers, suggesting **GA** conservative pruning approach potentially helped maintain the network's accuracy more effectively.
Overall, **GA** outperformed **SA** in terms of accuracy retention, with lower test losses across most layers.	

3. Sparsity and Complexity Reduction

Sparsity is a crucial metric in model pruning, as it represents the fraction of weights that are successfully pruned. Higher **sparsity** indicates a more lightweight model, leading to reduced computational requirements & memory consumption.

	Simulated Annealing (SA)	Genetic Algorithm (GA)
Over view	**SA** is effective at achieving significant sparsity without excessive complexity. It demonstrated solid performance in all layers, balancing pruning aggressiveness and loss. The method achieved sparsity levels between **13.33%** and **43.84%** across different layers. Hidden layers (such as **layers 5** and **6**) showed a much higher tolerance for pruning. This result is expected, as these fully connected layers often contain more redundant weights than convolutional layers.	**GA** achieved slightly lower sparsity, ranging between **11.20%** and **41.33%**. Although it managed to prune a significant number of weights, it was generally more conservative in its pruning approach, especially in the earlier layers (e.g., **layer 0**, where **GA** pruned only **11.2%** of the weights compared to **13.33%** in **SA**). It did achieve higher sparsity levels in some deeper layers. For example, in **layer 6**, it reached a sparsity of **38.59%**, which slightly outperforms **SA**'s result for that layer.
Pros	**SA** consistently reduced model complexity across layers while retaining performance. The gradual reduction in **temperature** ensures the pruning process is controlled and less likely to result in extreme configurations that hurt performance.	**GA** can explore more complex sparsity configurations, especially in deeper layers where larger pruning potential exists. Its evolutionary approach, driven by mutation and crossover, enables it to push sparsity levels higher than **SA** in some cases.
Cons	**SA** may not achieve the highest possible sparsity compared to **GA** in certain layers, as its deterministic nature limits the degree of exploration. While it achieves good results, its solutions may not always represent the global optimal in terms of sparsity.	While **GA** achieves higher sparsity in deeper layers, it may over-explore in earlier layers, leading to more iterations without substantial improvements. This is particularly true for layers where aggressive pruning may not be as beneficial, such as the input and shallow convolutional layers.
Overall, SA was more effective in maximizing sparsity, particularly in fully connected layers. Its higher overall sparsity of **43.84%** demonstrates its ability to more aggressively reduce model size.		

4. Accuracy and Performance

	Simulated Annealing (SA)	Genetic Algorithm (GA)
Pros	**SA** consistently maintained high accuracy even with increasing sparsity. For instance, after achieving a final sparsity of **13.33%** in **Layer 0**, the accuracy was **96.28%**. My **objective function** penalizes aggressive pruning that results in high loss, thus preserving accuracy.	**GA** maintained similar accuracy levels. It excels in maintaining accuracy while progressively improving sparsity, especially in deeper layers. Its population-based exploration allows it to avoid local minima better than **SA**, which can improve accuracy in more challenging layers.

Cons	While **SA** maintains accuracy well, it is somewhat limited by its deterministic nature again. There is less flexibility for large-scale exploration of the solution space, which restricts its effectiveness in finding globally optimal pruning masks, especially when model architecture becomes more complex	**GA** may sacrifice some performance in earlier layers due to its broader exploration, as seen in **Layer 0**, where the accuracy was **96.46%** (slightly higher than SA's **96.28%**), but sparsity was lower. Moreover, its longer execution time can limit its applicability in real-time or resource-constrained environments.

5. Execution Time

The computational efficiency of each algorithm is an important factor when considering large-scale models or deployment in production environments.

	Simulated Annealing (SA)	Genetic Algorithm (GA)
Overview	**SA** required **~11 mins 43 secs** to prune the **LeNet** model. This makes it a much faster option compared to **GA**.	**GA** took significantly longer to run, with a total execution time of **~27 mins 46 secs**. This is partly due to the genetic algorithm's need to maintain and evolve a population of pruning masks over multiple generations. Crossover and mutation operations, along with tournament selection, added substantial computational overhead.
Pros	**SA**'s fast execution time makes it a more practical option for large-scale models or real-time applications. It quickly converges on a solution and requires fewer resources compared to **GA**.	The additional time and computational cost result in a more thorough exploration of the solution space, which can be valuable when the goal is to discover the most optimal pruning configuration.
Cons	Faster execution time comes at the cost of less exploration, meaning that **SA** may miss certain pruning configurations that yield better sparsity-loss trade-offs in larger or more complex network	The extended execution time makes **GA** less practical in resource-constrained environments or when rapid pruning decisions are needed. It is also less suited for scenarios where real-time pruning is required.
Overall, **SA** was more computationally efficient than **GA** in this experiment. This significant time difference (**SA** is 2.37 times faster) is a crucial factor, especially when considering larger models or datasets. The efficiency of **SA** can be attributed to its single-solution approach, compared to **GA**'s population-based method.		

6. Adaptivity and Flexibility

Simulated Annealing (SA)	Genetic Algorithm (GA)
SA's flexibility comes from its ability to escape local minima by accepting suboptimal solutions early in the process. This makes it well-suited for highly non-convex problems, such as neural network pruning. However, the success of **SA** is highly dependent on carefully tuned parameters like the *cooling schedule* and *mutation rates*, which can make it less predictable in some cases. Despite this, it offers simplicity and ease of implementation, which can be advantageous in scenarios where quick deployment is necessary.	**GA**'s strength is its population-based approach, allowing it to explore a wider variety of pruning configurations. **GA** is also less sensitive to initial conditions compared to **SA**, as it can generate new solutions via *recombination*. Additionally, the use of **elitism** ensures good solutions are not lost during the evolution process, making **GA** more robust. However, population-based methods like **GA** could be a limitation for resource-constrained environments compared to **SA**, which maintains only 1 solution at a time, thus requiring less memory.

Recommendation of AI technique for the target problem

For the current problem of pruning a **LeNet-5** model on the **MNIST** dataset, **Simulated Annealing** is recommended as the most suitable technique. This is because:

- **SA**'s significantly faster runtime (2.37 times faster than **GA**) is a major advantage, especially when considering potential applications to larger models or datasets.
- **SA** achieved a slightly better trade-off between sparsity and accuracy, demonstrating its capability to find high-quality pruning configurations.
- **SA**'s quick initial improvement is beneficial for scenarios where fast results are needed.
- **SA**'s simpler implementation and fewer hyperparameters make it easier to deploy and fine-tune.

In general, while both techniques show promise, **SA** offers a more efficient and equally effective solution for the current neural network pruning task. However, if the goal is to explore the solution space more exhaustively and potentially achieve higher sparsity at the expense of time, **GA** could be a suitable alternative for complex or deeper networks.

Conclusions and Recommendations

This study explored the efficacy of 2 local search algorithms, **Simulated Annealing (SA)** and **Genetic Algorithm (GA)**, in pruning a simple pre-trained **LeNet** model on the **MNIST** dataset, focusing on reducing model sparsity by pruning unnecessary weights while maintaining its loss. Through experiments, this pruning is highly dependent on the network architecture, and fully connected layers tend to tolerate more pruning than convolutional layers. Here, both techniques were able to exploit this to varying degrees:

<table>
<tr><th colspan="2">Effectiveness of Both Techniques</th></tr>
<tr><td colspan="2">Both SA and GA demonstrated strong potential for reducing the number of weights in LeNet-5 model while maintaining acceptable loss and accuracy. SA achieved a sparsity of 43.84% with 87.44% accuracy, while GA reached 41.33% sparsity with 89.02% accuracy. As pruning increased, accuracy tended to decrease slightly, but both methods maintained reasonable accuracy even with higher sparsity levels.

Despite their significant time difference, both algorithms achieved comparable pruning results and exhibited the expected trade-off between sparsity and accuracy, with SA slightly edging out (trade-off scores of 0.3951 for SA vs. 0.3784 for GA). The cost functions for both methods converged, indicating their effectiveness in minimizing pruning-related penalties while maximizing model efficiency. These suggests that local search algorithms are indeed viable and effective for neural network pruning, addressing my primary research problem.</td></tr>
<tr><th>Performance of Simulated Annealing</th><th>Performance of Genetic Algorithm</th></tr>
<tr><td>SA demonstrated rapid initial improvement particularly in early layers of the network. It consistently converged faster than GA. In terms of time efficiency, it required less than half the execution time compared to GA, making it more practical and computationally efficient for time-sensitive applications or resource-constrained environments as it maintains only 1 solution at a time, requiring less memory and computation compared to population-based methods like GA.

However, SA's exploration of the solution space was more limited compared to GA, and it was less likely to find the global optimum in terms of sparsity, especially in deeper layers.

It also maintained a good balance between test loss and higher overall sparsity of 43.84%, indicating its ability to prune aggressively while preserving high accuracy. The temperature-based acceptance criterion allows SA to escape local optima early in the process. While effective, it could be highly sensitive to parameters such as the cooling schedule and mutation rates, which required careful tuning. The final solution can be more dependent on the initial state and cooling schedule, potentially missing global optima.</td><td>GA exhibited greater robustness in maintaining good solutions through its use of crossover, mutation, and elitism. This algorithm adapted well to the problem, ensuring the best-performing pruning masks were retained across generations. It can explore wider range of possible solutions simultaneously through its population-based approach. In addition to this, its ability to combine mutate solutions allowed GA to discover pruning masks that SA may not.

However, this came at the cost of significantly longer execution time. GA required over 27 minutes compared to SA's 11 minutes, making it less efficient for real-time or resource-constrained environments. While slower to converge, GA showed steady and more stable improvements with smoother transitions between layers, indicating its potential for finding more robust solutions given sufficient time. It was more conservative, achieving sparsity levels between 11.20% and 41.33% but still managed to prune a significant portion of the network's parameters.

With the broader exploration capabilities, it may be more suited for scenarios where maximizing sparsity while maintaining accuracy/loss is the primary goal, but the longer execution times can be tolerated. Its ability to discover better global solutions makes it particularly useful for deeper or more complex architectures. GA had more fine-tuned control over layer-specific pruning, especially in deeper layers where pruning decisions tend to be more complex.</td></tr>
<tr><td colspan="2"><u>Adaptability and Exploration</u>: Both algorithms implemented adaptive strategies to balance exploration and exploitation throughout the pruning process. Combined with the layer-wise pruning approach, both methods proved effective, suggesting good potential for scaling to larger, more complex networks.</td></tr>
</table>

Based on the findings and the characteristics of the 2 AI techniques, the following **<u>Recommendations</u>** can be made for practitioners and researchers looking to apply neural network pruning in various contexts:

- **Use Simulated Annealing for Fast, Moderate Pruning in Resource-Constrained Environments:**
  - For pruning tasks like the one presented in this study (medium-sized networks, moderate time constraints) or for scenarios where lightweight models are needed for deployment on devices with constrained memory and processing power, such as edge devices or mobile platforms, **SA** should be the preferred technique. Its faster convergence and lower computational requirements while maintaining high peformance makes it a practical choice when faster turnaround times are needed.

- **SA** is also a strong candidate for real-time pruning tasks, where model optimization needs to happen dynamically without incurring high computational overhead.

➢ **Use Genetic Algorithm for Large-Scale, Deep/Complex Networks:**

- In tasks where high sparsity is critical but with more complex datasets and deeper architectures such as ResNet or Transformer-based models, **GA** is recommended. **GA**'s evolutionary approach allows it to explore the solution space more thoroughly, ensuring that the final pruning masks are closer to the global optimum.
- **GA** is well-suited for offline model optimization where execution time is less of a concern, but maximizing sparsity is the key objective. It could be useful when dealing with architectures larger than **LeNet**, where more layers and a larger number of parameters allow for greater flexibility in pruning. Here, it can take advantage of its population-based exploration to find optimal pruning configurations.
- Additionally, parallelization techniques like Python Multi-Threading could be explored to mitigate **GA**'s computational intensity

➢ **Hybrid Approach for Fine-Tuning Pruning:**

Regarding this recommendation, I recommend 2 more different ways to conduct:

- **Simple way**: I recommend starting with **SA** first due to its efficiency. If computational resources allow and the pruning task proves challenging, switching to or comparing with **GA** might yield better results due to its population-based exploration.
- **Complex way**: Practitioners could consider combining both techniques in 1 run. For example, using **SA** for shallow layers to quickly achieve initial pruning results, followed by **GA** for fine-tuning on deeper layers, where more aggressive pruning is required, potentially leading to superior results. This could balance the strengths of both techniques, fast convergence in early stages and thorough exploration in later stages.

In conclusion, both **Simulated Annealing** and **Genetic Algorithm** offer effective solutions to the problem of neural network pruning, with each excelling under different conditions. Although both increase test losses, it may still be acceptable for less critical applications where maximum sparsity is the primary objective. Practitioners should carefully evaluate the needs of their specific application to select the most appropriate method. By further enabling the deployment of smaller, faster models without substantial performance loss, these methods contribute to the broader goal of making AI more accessible and practical across a wide range of applications and devices.

List of References

Aston, Z., Lipton, Z. C., Li, M., & Smola, A. J. (2023). Dive into Deep Learning. *arXiv.Org*. https://doi.org/10.48550/arxiv.2106.11342

Frankle, J., & Carbin, M. (2019). The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. *arXiv.Org*. https://doi.org/10.48550/arxiv.1803.03635

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Li, M., Sattar, Y., Thrampoulidis, C., & Samet Oymak. (2020). Exploring Weight Importance and Hessian Bias in Model Pruning. *arXiv.Org*. https://doi.org/10.48550/arxiv.2006.10903

Shachar, A. (2024). Introduction to Algogens. *arXiv.Org*. https://doi.org/10.48550/arxiv.2403.01426

Song, H., Pool, J., Tran, J., & Dally, W. J. (2015). Learning both Weights and Connections for Efficient Neural Networks. *arXiv.Org*. https://doi.org/10.48550/arxiv.1506.02626

Appendices

Appendix A: Cost function derivation

The objective function for each pruning configuration is defined as:

$$cost = loss\_diff + \frac{1}{sparsity + \varepsilon}, where:$$

- $loss\_diff = \begin{cases} (baseline\ loss - pruned\ loss)^2, & pruned\ loss < max\ loss \\ max\ loss\ penalty\ (10^8), & pruned\ loss \geq max\ loss \end{cases}$
- $sparsity = \frac{\sum_{l \in L_{prunable}} \sum_{i,j} 1\{W_l(i,j)=0\}}{\sum_{l \in L_{prunable}} Total\ weights\ in\ layer\ l}$

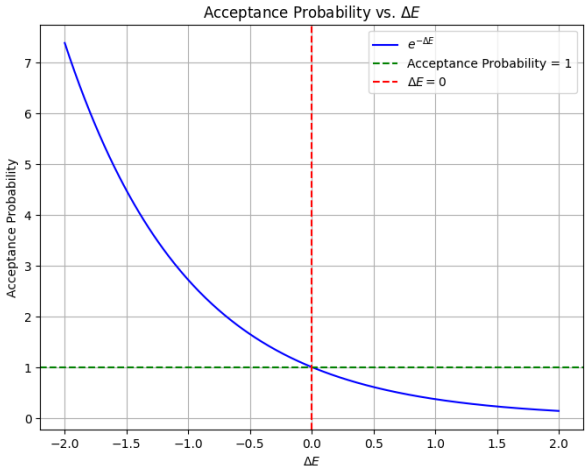
Appendix B: Acceptance Criterion in Simulated Annealing

$$P_{acceptance}(new\ solution) = \begin{cases} 1, & \Delta E < 0 \\ e^{\frac{-\Delta E}{T}}, & \Delta E \geq 0 \end{cases}, with\ \Delta E = cost_{new\ solution} - cost_{current\ solution}$$

This function is simply a function of e^{-x}. Therefore:

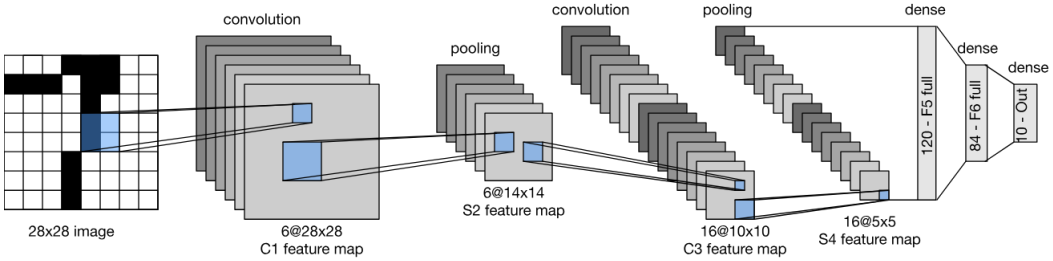
- If the change in the cost is negative, meaning new solution is better, the function will keep increasing to infinity.
- If the change is positive, the function will have a range of (0, 1], making it suitable for probability calculation.

```python
import numpy as np
import matplotlib.pyplot as plt

delta_E_values = np.linspace(-2, 2, 400)
acceptance_probabilities = np.exp(-delta_E_values)

plt.figure(figsize=(8, 6))
plt.plot(delta_E_values, acceptance_probabilities, label=r'$e^{-\Delta E}$'
plt.axhline(y=1, color='green', linestyle='--', label="Acceptance Probabili
plt.axvline(x=0, color='red', linestyle='--', label=r'$\Delta E = 0$')

plt.title("Acceptance Probability vs. $\Delta E$")
plt.xlabel(r'$\Delta E$')
plt.ylabel('Acceptance Probability')
plt.legend()
plt.grid(True)
plt.show()
```



Appendix C: Original LeNet architecture

There are many implementations of the **LeNet** architecture available online, but only a few closely follow the original design outlined in the paper by LeCun et al. (1998). The implementation used in this project is based on the architecture presented in the book **Dive into Deep Learning** by Aston et al. (2023), which is the most similar to the original LeNet:



This implementation captures the key features of LeCun et al. (1998)'s architecture, including the use of **Average Pooling** layers and **Sigmoid** activation functions, which are often replaced by **Max Pooling** and **ReLU** activations in other implementations.